REPORT DOCUMENTATION PAGE

AD-A244 847

1. AGE	DATE	3. REPORT TYPE AND DATES COVERED
		Final: 18 Dec 1990 to 01 Jun 1993

4. TITLE AND SUBTITLE

Alliant Computer Systems Corporation, Alliant FX/Ada Compiler, Version 2.3, Alliant FX/80 (Host & Target), 901218W1.11106

5. FUNDING NUMBERS

6. AUTHOR(S)

Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB, Dayton, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER

AVF-VSR-443.1191

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Alliant Computer Systems Corporation, Alliant FX/Ada Compiler, Wright-Patterson, AFB, Version 2.3, Alliant FX/80 under Concentrix, Release 5.7, (Host & Target), ACVC 1.11.

DTIC
S ELECTE D
JAN 13 1992
B

14. SUBJECT TERMS

Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901218W1.11106
Alliant Computer Systems Corporation
Alliant FX/Ada Compiler, Version 2.3
Alliant FX/80 => Alliant FX/80

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

92-01048

02 4 10 039

Certificate Information

The following Ada implementation was tested and determined to pass ACVC
1.11. Testing was completed on 18 December 1990.

Compiler Name and Version: Alliant FX/Ada Compiler, Version 2.3

Host Computer System: Alliant FX/80 under Concentrix, Release 5.7

Target Computer System: Alliant FX/80 under Concentrix, Release 5.7

Customer Agreement Number: 90-10-18-ACS

A more detailed description of this Ada implementation is found in section
3.1 of this report.

As a result of this validation effort, Validation Certificate
901218W1.11106 is awarded to Alliant Computer Systems Corporation. This
certificate expires on 1 March 1993.

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

DECLARATION OF CONFORMANCE

Compiler Implementor: Alliant Computer Systems Corporation
Ada Validtaion Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability Version: 1.11

Base Configuration

Base Compiler Name: Alliant FX/Ada Compiler, Version 2.3
Host   Architecture ISA: Alliant FX/80 OS&Ver#: Concentrix Release 5.7
Target Architecture ISA: Alliant FX/80 OS&Ver#: Concentrix Release 5.7

Implementor's Declaration

I, the undersigned, representing Alliant, have implemented
no deliberate extensions to the Ada Language Standard
ANSI/MIL-STD-1815A in the compilers listed in this declara-
tion. I declare that Alliant Computer Systems Corporation is
the owner of record of the Ada language compilers listed
above and, as such, is responsible for maintaining said com-
pilers in conformance to ANSI/MIL-STD-1815A.   All certifi-
cates and registrations for Ada language compilers listed in
the declaration shall be made only in the owner's corporate
name.

Andrew F. Halford   Date: _Dec. 7, 1990_
Alliant Computer Systems Corporation
Andrew F. Halford

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

> National Technical Information Service
> 5285 Port Royal Road
> Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

> Ada Validation Organization
> Institute for Defense Analyses
> 1801 North Beauregard Street
> Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language,
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program
Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC
contains a collection of test programs structured into six test classes:
A, B, C, D, E, and L. The first letter of a test name identifies the class
to which it belongs. Class A, C, D, and E tests are executable. Class B
and class L tests are expected to produce errors at compile time and link
time, respectively.

The executable tests are written in a self-checking manner and produce a
PASSED, FAILED, or NOT APPLICABLE message indicating the result when they
are executed. Three Ada library units, the packages REPORT and SPPRT13,
and the procedure CHECK_FILE are used for this purpose. The package REPORT
also provides a set of identity functions used to defeat some compiler
optimizations allowed by the Ada Standard that would circumvent a test
objective. The package SPPRT13 is used by many tests for Chapter 13 of the
Ada Standard. The procedure CHECK_FILE is used to check the contents of
text files written by some of the Class C tests for Chapter 14 of the Ada
Standard. The operation of REPORT and CHECK_FILE is checked by a set of
executable tests. If these units are not operating correctly, validation
testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class
B tests are not executable. Each test in this class is compiled and the
resulting compilation listing is examined to verify that all violations of
the Ada Standard are detected. Some of the class B tests contain legal Ada
code which must not be flagged illegal by the compiler. This behavior is
also verified.

Class L tests check that an Ada implementation correctly detects violation
of the Ada Standard involving multiple, separately compiled units. Errors
are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by
implementation-specific values -- for example, the largest integer. A list
of the values used for this implementation is provided in Appendix A. In
addition to these anticipated test modifications, additional changes may be
required to remove unforeseen conflicts between the tests and
implementation-dependent characteristics. The modifications required for
this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the
AVF. This customization consists of making the modifications described in
the preceding paragraph, removing withdrawn tests (see section 2.1) and,
possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of
the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler   The software and any needed hardware that have to be added
                   to a given host and target computer system to allow
                   transformation of Ada programs into executable form and
                   execution thereof.

Ada Compiler   The means for testing compliance of Ada implementations,
Validation   consisting of the test suite, the support programs, the ACVC
Capability   user's guide and the template for the validation summary
(ACVC)   report.

Ada   An Ada compiler with its host computer system and its
Implementation   target computer system.

Ada Joint   The part of the certification body which provides policy and
Program   guidance for the Ada certification system.
Office (AJPO)

Ada   The part of the certification body which carries out the
Validation   procedures required to establish the compliance of an Ada
Facility (AVF)   implementation.

Ada   The part of the certification body that provides technical
Validation   guidance for operations of the Ada certification system.
Organization
(AVO)

Compliance of   The ability of the implementation to pass an ACVC version.
an Ada
Implementation

Computer   A functional unit, consisting of one or more computers and
System   associated software, that uses common storage for all or
                   part of a program and also for all or part of the data
                   necessary for the execution of the program;   executes
                   user-written   or   user-designated   programs;   performs
                   user-designated data manipulation, including arithmetic
                   operations and logic operations; and that can execute
                   programs that modify themselves during execution.   A
                   computer system may be a stand-alone unit or may consist of
                   several inter-connected units.

Conformity   Fulfillment by a product, process or service of all
                  requirements specified.

Customer   An individual or corporate entity who enters into an
                  agreement with an AVF which specifies the terms and
                  conditions for AVF services (of any kind) to be performed.

Declaration of   A formal statement from a customer assuring that conformity
Conformance   is realized or attainable on the Ada implementation for
                  which validation status is realized.

Host Computer   A computer system where Ada source programs are transformed
System   into executable form.

Inapplicable   A test that contains one or more test objectives found to be
test   irrelevant for the given Ada implementation.

ISO   International Organization for Standardization.

LRM   The Ada standard, or Language Reference Manual, published as
                  ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from
                  the LRM take the form "<section>.<subsection>:<paragraph>."

Operating   Software that controls the execution of programs and that
System   provides services such as resource allocation, scheduling,
                  input/output control, and data management. Usually,
                  operating systems are predominantly software, but partial or
                  complete hardware implementations are possible.

Target   A computer system where the executable form of Ada programs
Computer   are executed.
System

Validated Ada   The compiler of a validated Ada implementation.
Compiler

Validated Ada   An Ada implementation that has been validated successfully
Implementation   either by AVF testing or by registration [Pro90].

Validation   The process of checking the conformity of an Ada compiler to
                  the Ada programming language and of issuing a certificate
                  for this implementation.

Withdrawn   A test found to be incorrect and not used in conformity
test   testing. A test may be incorrect because it has an invalid
                  test objective, fails to meet its test objective, or
                  contains erroneous or illegal use of the Ada programming
                  language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for
withdrawing each test is available from either the AVO or the AVF. The
publication date for this list of withdrawn tests is 21 November 1990.

E28005C	B28006C	C34006D	C35702A	B41308B	C43004A
C45114A	C45346A	C45612B	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	B85001L	C83026A	C83041A	C97116A	C98003B
BA2011A	CB7001A	CB7001B	CB7004A	CC1223A	BC1226A
CC1226B	BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A
CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A
CD2B15C	BD3006A	BD4008A	CD4022A	CD4022D	CD4024B
CD4024C	CD4024D	CD4031A	CD4051D	CD5111A	CD7004C
ED7005D	CD7005E	AD7006A	CD7006E	AD7201A	AD7201E
CD7204B	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant
for a given Ada implementation. The inapplicability criteria for some
tests are expla_ned in documents issued by ISO and the AJPO known as Ada
Issues and commonly referenced in the format AI-dddd. For this
implementation, the following tests were inapplicable for the reasons
indicated; references to Ada Issues are included as appropriate.

IMPLEMENTATION DEPENDENCIES

The following 201 tests nave floating-point type declarations requiring more digits than SYSTEM.MAX_DIG "S:

```
C24113L..Y (14 tests)      C35705L..Y (14 tests)
C35706L..Y (14 tests)      C35707L..Y (14 tests)
C35708L..Y (14 tests)      C35802L..Z (15 tests)
C45241L..Y (14 tests)      C45321L..Y (14 tests)
C45421L..Y (14 tests)      C45521L..Z (15 tests)
C45524L..Z (15 tests)      C45621L..Z (15 tests)
C45641L..Y (14 tests)      C46012L..Z (15 tests)
```

The following 21 tests check for the predefined type LONG_INTEGER:

```
C35404C     C45231C     C45304C     C45411C     C45412C
C45502C     C45503C     C45504C     C45504F     C45611C
C45612C     C45613C     C45614C     C45631C     C45632C
B52004D     C55B07A     B55B09C     B86001W     C86006C
CD7101F
```

C35702B, C35713C, B86001U, and C86006G check for the predefined type LONG_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

A35801E checks that FLOAT'FIRST..FLOAT'LAST may be used as a range constraint in a floating-point type declaration; for this implementation, that range exceeds the range of safe numbers of the largest predefined floating-point type and must be rejected. (See section 2.3.)

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater.

C45624A checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 5. For this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

C86001F recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete. For this implementation, the package TEXT_IO is dependent upon package SYSTEM.

B86001Y checks for a predefined fixed-point type other than DURATION.

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

IMPLEMENTATION DEPENDENCIES

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 14 tests.

The following tests were split into two or more tests because this
implementation did not report the violations of the Ada Standard in the way
expected by the original tests.

B24009A	B33301B	B38003A	B38003B	B38009A	B38009B
B85008G	B85008H	BC1303F	BC3005B	BD2B03A	BD2D03A
BD4003A					

A35801E was graded inapplicable by Evaluation Modification as directed by
the AVO. The compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST
as the range constraint of a floating-point type declaration because the
bounds lie outside of the range of safe numbers (cf. LRM 3.5.7:12).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described
adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada
implementation system, see:

> Alliant Computer Systems Corporation
> One Monarch Drive
> Littleton MA 01460

For a point of contact for sales information about this Ada implementation
system, see:

> Alliant Computer Systems Corporation
> One Monarch Drive
> Littleton MA 01460

Testing of this Ada implementation was conducted at the customer's site by
a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test
of the customized test suite in accordance with the Ada Programming
Language Standard, whether the test is applicable or inapplicable;
otherwise, the Ada Implementation fails the ACVC [Pro90].

PROCESSING INFORMATION

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a)	Total Number of Applicable Tests	3814
b)	Total Number of Withdrawn Tests	83
c)	Processed Inapplicable Tests	72
d)	Non-Processed I/O Tests	0
e)	Non-Processed Floating-Point Precision Tests	201
f)	Total Number of Inapplicable Tests	273
g)	Total Number of Tests for ACVC 1.11	4170

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 273 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

    -w   suppress warnings

    -M   link the test into an executable image

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC.
The meaning and purpose of these parameters are explained in [UG89]. The
following macro parameters are defined in terms of the value V of
$MAX_IN_LEN which is the maximum input line length permitted for the tested
implementation. For these parameters, Ada string expressions are given
rather than the macro values themselves.

Macro Parameter	Macro Value
$MAX_IN_LEN	499
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"

MACRO PARAMETERS

$MAX_STRING_LITERAL   '"' & (1..V-2 => 'A') & '"'

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_647
$DEFAULT_MEM_SIZE	16_777_216
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	FX_UNIX
$DELTA_DOC	0.00000000046566128730773392578125
$ENTRY_ADDRESS	SYSTEM.PHYSICAL_ADDRESS(16#40#)
$ENTRY_ADDRESS1	SYSTEM.PHYSICAL_ADDRESS(16#80#)
$ENTRY_ADDRESS2	SYSTEM.PHYSICAL_ADDRESS(16#100#)
$FIELD_LAST	2_147_483_647
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	10_000_000.0
$GREATER_THAN_FLOAT_BASE_LAST	1.8E+308
$GREATER_THAN_FLOAT_SAFE_LARGE	5.0E307
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	

```
                       9.0E37

$HIGH_PRIORITY         99

$ILLEGAL_EXTERNAL_FILE_NAME1
                       7illegal/file_name/2}]%2102c.dat

$ILLEGAL_EXTERNAL_FILE_NAME2
                       7illegal/file_name/CE2102C.dat

$INAPPROPRIATE_LINE_LENGTH
                       -1

$INAPPROPRIATE_PAGE_LENGTH
                       -1

$INCLUDE_PRAGMA1       PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2       PRAGMA INCLUDE ("B28006D1.TST")

$INTEGER_FIRST         -2_147_483_648

$INTEGER_LAST          2_147_483_647

$INTEGER_LAST_PLUS_1   2_147_483_648

$INTERFACE_LANGUAGE    C

$LESS_THAN_DURATION    -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                       -10_000_000.0

$LINE_TERMINATOR       ASCII.LF

$LOW_PRIORITY          0

$MACHINE_CODE_STATEMENT
                       CODE_0'(OP => NOP);

$MACHINE_CODE_TYPE     CODE_0

$MANTISSA_DOC          31

$MAX_DIGITS            15

$MAX_INT               2_147_483_647

$MAX_INT_PLUS_1        2_147_483_648

$MIN_INT               -2_147_483_648

$NAME                  TINY_INTEGER
```

MACRO PARAMETERS

$NAME_LIST	FX_UNIX
$NAME_SPECIFICATION1	/lang3/ada/acvc1_11/c/e/X2120A
$NAME_SPECIFICATION2	/lang3/ada/acvc1_11/c/e/X2120B
$NAME_SPECIFICATION3	/lang3/ada/acvc1_11/c/e/X3119A
$NEG_BASED_INT	16#FFFFFFFD#
$NEW_MEM_SIZE	16_777_216
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	FX_UNIX
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	RECORD SUBP : OPERAND; END RECORD;
$RECORD_NAME	CODE_0
$TASK_SIZE	32
$TASK_STORAGE_SIZE	10240
$TICK	0.01
$VARIABLE_ADDRESS	VAR_1'ADDRESS
$VARIABLE_ADDRESS1	VAR_2'ADDRESS
$VARIABLE_ADDRESS2	VAR_3'ADDRESS
$YOUR_PRAGMA	RESOURCE

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

NAME

   ada - FX/Ada compiler

SYNOPSIS

   ada [options] [ada_source.a ...] [ld_options] [object_file.o]...

DESCRIPTION

   Compiles the specified Ada source files into object files and
   optionally (if -M is specified) links the object files into
   an executable file. Object files from previous Ada
   compilations are implicitly included as needed. Object files
   from non-Ada compilations can be explicitly included by
   specifying the names of the .o files.

   Source files must reside in directories that are FX/ADS
   libraries (see a.mklib); source file names must end in .a (a
   period followed by the letter a in lowercase). Output object
   files are placed in the invisible subdirectory .objects and
   are not visible as .o files in the FX/ADS library. Separate
   compilation information is placed in the invisible
   subdirectory .nets.

   If error processing (-e, -E, -el, -El, or -ev) is not
   specified, the raw error messages are written to standard
   error output. If you want to process the raw error messages
   with e.error, redirect standard error output to a file (>&
   filename at the end of the command line).

OPTIONS

   -a file_name
        (archive) Treats file_name.a as an archive file rather
        than an Ada source file.

   -d   (dependencies) Analyze the source file for dependencies
        only. No syntax analysis occurs and no object files are
        produced. Used by a.make to establish dependencies
        among new files.

   -e   (error) Writes error lines and diagnostics to standard
        output. Only one of the -e or -E options should be used.

   -E [file|directory]
        (error) Writes error lines and diagnostics to standard
        output and writes the raw error messages to the
        specified file. The name of the file defaults to
        ada_source.err; if a directory name is specified, the
        output is placed in ada_source.err in that directory.
        Only one of the -e or -E options should be used.

-el (error listing) Writes a full listing with interspersed diagnostics to standard output, if any errors occur. Only one of the -el or -El options should be used.

-El [file|directory]
    (error listing) Writes a full listing with interspersed diagnostics to standard output and writes the raw error messages to the specified file, if any errors occur. The name of the file defaults to ada_source.err; if a directory name is specified, the output is placed in ada_source.err in that directory. Only one of the -el or -El options should be used.

-ev (error vi) Embeds the raw error messages in the source file and calls vi on the source file.

-lx (link) Includes the library libx.a from /lib, /usr/lib, or /usr/local/lib. This option is a link option and\ must not precede the name of a file that references the library. See the Concentrix ld command.

-M unit_name
    (main) Produces an executable program using the named program unit as the main program. The main program must be either a parameterless procedure or a parameterless function returning an integer.

-M ada_source.a
    (main) Like -M unit_name, except that the unit name is assumed to be the root name of the source file that follows

-o executable_file
    (output) Names the output executable file; by default, the executable file is named a.out. The -M option must also be specified.

-O[n]
    (optimize) Optimizes the output code. An optional digit limits the number of optimization passes; 9 specifies maximum optimization. The default number of optimization passes is 1.

-pg Produces a program that (at program execution time) monitors the calling of routines and writes a gmon.out file. The Concentrix command gprof -A processes this file.

-R library
    (recompile instantiation) Forces an analysis of all generic instantiations, causing reinstantiation of any that are out of date.

-S   (suppress) Applies the pragma SUPPRESS to the entire compilation.

-sh   (show) Shows the pathname of the tool actually called.

-T   (timing) Prints timing information for the compilation.

-u   (update) Updates the library ada.lib even if syntax errors are present.

-v   (verbose) Prints the compiler version number, the date and time of compilation, and summary information concerning the compilation.

-w   (warnings) Suppresses warning diagnostics.

EXAMPLES

The following example compiles the source file hello.a

    ada hello.a

The following example compiles the source file hello.a and produces an executable program named a.out. The main program is the program unit named hello.

    ada -M hello.a

The following example compiles the source files termspec.a, termbody.a, and hanoi.a, and produces an executable program named a.out. The main program is the program unit named hanoi.

    ada -M hanoi termspec.a termbody.a hanoi.a

The following example compiles the source file hello.a and produces an executable program named hello. The main program is the program unit named hello.

    ada -o hello -M hello.a

The following example writes error lines and diagnostics to standard output.

    ada -e -o hello -M hello.a

The following example writes a listing with interspersed diagnostics to standard output and the raw error messages to hello.err.

    ada -El -o hello -M hello.a

FILES

    FILE.A - Ada source input file
    /tmp/file.$$ - code file created by front end
    ada.lib - FX/Ads directory information file
    gnrx.lib - FX/Ads generics library information file
    GVAS_table - GVAS table in the current FX/ADA project
    ada.lock - lock link to ada.lib, for mutual exclusion
    GVAS_table.LOCK - lock   link   to   GVAS_table,   for   mutual
       exclusion

SEE ALSO
    a.das, a.error, a.ld, a.mklib, ld(1)

DIAGNOSTICS

The diagnostics produced by the VADS compiler are intended
to be self-explanatory. Most refer to the RM. Each RM
reference includes a section number and optionally, a
paragraph number enclosed in parentheses.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to
implementation-dependent pragmas, to certain machine-dependent conventions
as mentioned in Chapter 13 of the Ada Standard, and to certain allowed
restrictions on representation clauses. The implementation-dependent
characteristics of this Ada implementation, as described in this Appendix,
are provided by the customer. Unless specifically noted otherwise,
references in this Appendix are to compiler documentation and not to this
report. Implementation-specific portions of the package STANDARD, which
are not a part of Appendix F, are:

```
package STANDARD is
    ..........
    type INTEGER is range -2147483648 .. 2147483647;
    type SHORT_INTEGER is range -32768 .. 32767;
    type TINY_INTEGER is range -128 .. 127;

    type FLOAT is digits 15 range
        -1.79769313486235E+308 .. +1.79769313486235E+308;
    type SHORT_FLOAT is digits 6
        range -3.40283E+38 .. 3.40283E+38;

    type DURATION is delta 0.001 range -2147483.648 .. 2147483.647;
    ..........
end STANDARD;
```

ATTACHMENT V

APPENDIX F. Implementation-Dependent Characteristics

FX/Ada release 2.3 Compiler

1. "Implementation-Dependent Pragmas"

INLINE_ONLY Pragma

The INLINE_ONLY pragma, when used in the same way as pragma INLINE, indicates to the compiler that the sub-program must lways be inlined. This pragma also suppresses the generation of a callable version of the routine which saves code space.

BUILT_IN Pragma

The BUILT_IN pragma is used in the implementation of some predefined Ada packages, but provides no user access. It is used only to implement code bodies for which no actual Ada body can be provided, for example the MACHINE_CODE package.

RESOURCE Pragma

The RESOURCE pragma specifies the resource class of the task (or tasks of a task type) or the resource class of the main program. It is used to force a task to execute on a particular resource class when multiple processors are used to execute an Ada program. This pragma takes a static expression of the type RESOURCE_TYPE declared in package SYSTEM. This pragma is only allowed within the specification of a task unit or immediately within the outermost declarative part of a main program.

SHARE_CODE Pragma

The SHARE_CODE pragma takes the name of a generic instantiation or a generic unit as the first argument and one of the identifiers TRUE or FALSE as the second argument. This pragma is only allowed immediately at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

EXTERNAL_NAME Pragma

The EXTERNAL_NAME pragma takes the name of a subprogram
or variable defined in Ada and allows the user to
specify an external name that may be used to reference
the entity from other languages. The pragma is allowed
at the place of a declarative item in a package specif-
ication and must apply to an object declared earlier in
the same package specification.

INTERFACE_NAME Pragma

The INTERFACE_NAME pragma takes the name of a a vari-
able defined in another language and allows it to be
referenced directly in Ada. The pragma will replace
all occurrences of the variable name with an external
reference to the second, link_argument. The pragma is
allowed at the place of a declarative item in a package
specification and must apply to an object declared ear-
lier in the same package specification. The object
must be declared as a scalar or an access type. The
object cannot be any of the following:
        a loop variable,
        a constant,
        an initialized variable,
        an array, or
        a record.
The INTERFACE_NAME pragma is also used to provide an
external link name to a subprogram which has been
defined using the INTERFACE pragma.

IMPLICIT_CODE Pragma

Takes one of the identifiers ON or OFF as the single
argument. This pragma is only allowed within a machine
code procedure. It specifies that implicit code gen-
erated by the compiler be allowed or disallowed. A
warning is issued if OFF is used and any implicit code
needs to be generated. The default is ON.

LINK_WITH Pragma

The LINK_WITH pragma specifies a command to add to the
a.ld link line. This pragma takes one argument, a
string.

NON_REENTRANT

This pragma takes one argument which can be the name of
either a library subprogram or a subprogram declared
immediately within a library package spec or body. It
indicates to the compiler that the subprogram will not
be called recursively allowing the compiler to perform
specific optimizations. The pragma can be applied to a

subprogram or a set of overloaded subprogram within a package spec or package body.

NOT_ELABORATED

This pragma can only appear in a library package specification. It indicates that the package will not be elaborated because it is either part of the RTS, a configuration package or an Ada package that is referenced from a language other than Ada. The presence of this pragma suppresses the generation of elaboration code and issues warnings if elaboration code is required.

2. Implementation of Predefined Pragmas

CONTROLLED

This pragma is recognized by the implementation but has no effect.

ELABORATE

This pragma is implemented as described in Appendix B of the Ada RM.

INLINE
This pragma is implemented as described in Appendix B of the Ada RM.

INTERFACE

This pragma supports calls to 'C', Pascal, and FORTRAN functions. The Ada subprograms can be either functions or procedures. The types of parameters and the result type for functions must be scalar, access or the predefined type ADDRESS in SYSTEM. All parameters must have mode IN. Record and array objects can be passed by reference using the ADDRESS attribute.

LIST

This pragma is implemented as described in Appendix B of the Ada RM.

MEMORY_SIZE

This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

OPTIMIZE

> This pragma is recognized by the implementation but has no effect.

PACK

> This pragma will cause the compiler to choose a non-aligned representation for composite types. It will not causes objects to be packed at the bit level.

PAGE

> This pragma is implemented as described in Appendix B of the Ada RM.

PRIORITY

> This pragma is implemented as described in Appendix B of the Ada RM.

SHARED

> This pragma is recognized by the implementation but has no effect.

STORAGE_UNIT

> This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

SUPPRESS

> This pragma is implemented as described, except that RANGE_CHECK and DIVISION_CHECK cannot be supressed.

SYSTEM_NAME

> This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

3. Implementation-Dependent Attributes

P'REFREF'u>(120u+1n) .br For a prefix that denotes an object, a program unit, a label, or an entry.

> This attribute denotes the effective address of the

first of the storage units allocated to P. For a sub-program, package, task unit, or label, it refers to the address of the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, it refers to the corresponding hardware interrupt. The attribute is of the type OPERAND defined in the package MACHINE_CODE. The attribute is only allowed within a machine code procedure.

See section F.4.8 for more information on the use of this attribute.

(For a package, task unit, or entry, the 'REF attribute is not supported.)

4.   Values of Predefined Attributes

Attributes of the pre-defined type DURATION
    (a fixed-point type)

first is -2147483.648
last is   2147483.647
size is   32
delta is   1.00000000000000E-03
mantissa is   31
small is   1.00000000000000E-03
large is   2.14748364700000E+06
fore is   8
aft is   3
safe_small is   1.00000000000000E-03
safe_large is   2.14748364700000E+06
machine_rounds is TRUE
machine_overflows is TRUE

Attributes of type FLOAT
size   64
first   -1.79769313486235E+308
last   1.79769313486235E+308
digits   15
mantissa   51
epsilon   8.88178419700125E-16
emax   204
small   1.94469227433161E-62
large   2.57110087081439E+61
safe_emax   1021
safe_small   2.22507385850720E-308
safe_large   2.24711641857793E+307
machine_radix   2
machine_mantissa   53

```
machine_emax              1024
machine_emin              -1021
machine_rounds       TRUE
machine_overflows        TRUE
```

Attributes of type SHORT_FLOAT
```
size              32
first       -3.40283E+38
last         3.40283E+38
digits               6
mantissa             21
epsilon       9.53674316406250E-07
emax             84
small        2.58493941422821E-26
large        1.93428038904621E+25
safe_emax             125
safe_small      1.17549435082229E-38
safe_large      4.25352755827078E+37
machine_radix              2
machine_mantissa              23
machine_emax              128
machine_emin              -125
machine_rounds       TRUE
machine_overflows        TRUE
```

Ranges of predefined integer types

```
TINY_INTEGER
-128 ..   127
SHORT_INTEGER
-32768 ..   32767
INTEGER
-2147483648 ..   2147483647
```

Default STORAGE_SIZE (collection size) for an access type

     100000

Priority range is 0 .. 99

Default Storage Size for Tasks is
      20480
If tasks need larger stack sizes, the 'STORAGE_SIZE attribute
may be used with the task type declaration.

Attributes and time-related numbers

```
Duration'small 1.00000000000000E-03
System.tick 1.00000000000000E-02
```

5.   Specification Of Package SYSTEM


```
package SYSTEM
is
     type NAME is ( fx_unix );

     SYSTEM_NAME: constant NAME := fx_unix;

     STORAGE_UNIT: constant := 8;
     MEMORY_SIZE: constant := 16_777_216;

     -- System-Dependent Named Numbers

     MIN_INT: constant := -2_147_483_648;
     MAX_INT: constant := 2_147_483_647;
     MAX_DIGITS: constant := 15;
     MAX_MANTISSA: constant := 31;
     FINE_DELTA: constant := 2.0**(-31);
     TICK: constant := 0.01;

     -- Other System-dependent Declarations

     subtype PRIORITY is INTEGER range 0 .. 99;
     MIN_PRIORITY: constant priority := priority'first;
     MAX_PRIORITY: constant priority := priority'last;

     type RESOURCE_TYPE is (any_resource,
            detached_ce_resource,
            complex_resource,
            ip_resource);

     unavailable_resource : exception;

     MAX_REC_SIZE : integer := 64*1024;

     type ADDRESS is private;

     NO_ADDR : constant ADDRESS;

     function PHYSICAL_ADDRESS(I: INTEGER) return ADDRESS;
     function ADDR_GT(A, B: ADDRESS) return BOOLEAN;
     function ADDR_LT(A, B: ADDRESS) return BOOLEAN;
     function ADDR_GE(A, B: ADDRESS) return BOOLEAN;
     function ADDR_LE(A, B: ADDRESS) return BOOLEAN;
     function ADDR_DIFF(A, B: ADDRESS) return INTEGER;
     function INCR_ADDR(A: ADDRESS; INCR: INTEGER) return ADDRESS;
     function DECR_ADDR(A: ADDRESS; DECR: INTEGER) return ADDRESS;

     function ">" (A, B: ADDRESS) return BOOLEAN renames ADDR_GT;
     function "<" (A, B: ADDRESS) return BOOLEAN renames ADDR_LT;
     function ">="(A, B: ADDRESS) return BOOLEAN renames ADDR_GE;
     function "<="(A, B: ADDRESS) return BOOLEAN renames ADDR_LE;
     function "-" (A, B: ADDRESS) return INTEGER renames ADDR_DIFF;
```

```
function "+" (A: ADDRESS;
                 INCR: INTEGER) return ADDRESS renames INCR_ADDR;
function "-"(A: ADDRESS;
                 DECR: INTEGER) return ADDRESS renames DECR_ADDR;

    pragma inline(ADDR_GT);
    pragma inline(ADDR_LT);
    pragma inline(ADDR_GE);
    pragma inline(ADDR_LE);
    pragma inline(ADDR_DIFF);
    pragma inline(INCR_ADDR);
    pragma inline(DECR_ADDR);
    pragma inline(PHYSICAL_ADDRESS);

private

    type ADDRESS is new integer;

    NO_ADDR : constant ADDRESS := 0;

end SYSTEM;
```

6.   Restrictions On Representation Clauses

Pragma PACK

In the absence of pragma PACK record components are
padded so as to provide for efficient access by the
target hardware. Pragma PACK applied to a record elim-
inates the padding where possible. Pragma PACK has no
other effect on the storage allocated for record com-
ponents, so a record representation is required to make
record components smaller. Bit packing is not sup-
ported for components larger than STORAGE_UNIT. Com-
ponents smaller than STORAGE_UNIT will be bit packed
within a storage unit. Objects and larger components
are packed to the nearest whole STORAGE_UNIT.

Length Clauses

For scalar types, a length clause which is a size
specification will compress storage to the number of
bits required to represent the range of the subtype.
For   fixed, short_float, and access types, this is 32.
For float, this is 64.

A size specification applied to a composite type with
components of composite types will not cause compres-
sion of component storage.   To   allocate   the minimal
number of bits for records of composite types, an
explicit record representation clause must be given

with length clauses for each component. An error will
be issued if there is insufficient space allocated.
Component clauses need not be aligned on STORAGE_UNIT
boundaries. A component of a record representation
clause may not specify fewer bits for a component type
than would be used for values of the type.

Size specifications (T'SIZE) are not supported for task
types. Specifications of storage for a task activation
(T'STORAGE_SIZE) is supported. The minimum storage
size for task activations is 5120, but can be larger
depending on the size of data objects declared in the
task.

The size specification T'SMALL is not supported except
when the representation specification is the same as
the value 'SMALL for the base type.

Specification of collection size is supported.

Address Clauses

Address clauses are supported for uninitialized vari-
ables and constants. They are not supported by the
compiler for subprograms, packages, and task units.

Interrupts

Interrupt entries are supported for UNIX signals. The Ada for clause gives the UNIX signal number. The following is the meaning associated with the valid UNIX signals:

1	SIGHUP	hangup
2	SIGINT	interrupt
3	SIGQUIT	quit
4	SIGILL	illegal instruction (not reset when caught)
5	SIGTRAP	trace trap (not reset when caught)
6	SIGIOT	IOT instruction
7	SIGEMT	EMT instruction
8	SIGFPE	floating point exception
9	SIGKILL	kill (cannot be caught or ignored)
10	SIGBUS	bus error
11	SIGSEGV	segmentation violation
12	SIGSYS	bad argument to system call
13	SIGPIPE	write on a pipe with no one to read it
14	SIGALRM	alarm clock
15	SIGTERM	software termination signal from kill
16	SIGURG	urgent condition on IO channel
17	SIGSTOP	sendable stop signal not from tty
18	SIGTSTP	stop signal from tty
19	SIGCONT	continue a stopped process
20	SIGCHLD	to parent on child stop or exit
21	SIGTTIN	to readers pgrp upon background tty read
22	SIGTTOU	like TTIN for output if (tp->t_local&LTOSTOP)
23	SIGIO	input/output possible signal
24	SIGXCPU	exceeded CPU time limit
25	SIGXFSZ	exceeded file size limit
26	SIGVTALRM	virtual time alarm
27	SIGPROF	profiling time alarm
28	SIGWINCH	window changed

Representation Attributes

The ADDRESS attribute is not supported for packages and task entries. The compiler issues a warning message and the value which is type SYSTEM.ADDRESS is SYSTEM.NO_ADDR.

Machine Code Insertions

Machine code insertions are supported.

The general definition of the package MACHINE_CODE provides an assembly language interface for the target machine. It provides the necessary record type(s) needed in the code statement, an enumeration type of all the opcode mneumonics, a set of register definitions, and a set of addressing mode functions.

The general syntax of a machine code statement is as follows:

CODE_n' (opcode, operand {, operand});

where n indicates the number of operands in the aggregate.

A special case arises for a variable number of operands. The operands are listed within a subaggregate. The format is as follows:

CODE_N' (opcode, (operand {, operand}));

For those opcodes that require no operands, named notation must be used (cf. RM 4.3(4)).

CODE_0' (op => opcode);

The opcode must be an enumeration literal (i.e. it cannot be an object, attribute, or a rename).

An operand can only be an entity defined in MACHINE_CODE or the 'REF attribute.

The arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE. The 'REF attribute may not be used as an argument in any of these functions.

Inline expansion of machine code procedures is supported.

7. Conventions for Implementation-generated Names

There are no implementation-generated names.

8. Interpretation of Expressions in Address Clauses

Address clauses are supported for constants and variables. Address clauses for interrupts are interpreted as described above.

9. Restrictions on Unchecked Conversions

None.

10. Restrictions on Unchecked Deallocations

None.

11. Implementation Characteristics of I/O Packages

The input output packages are implemented as specified in Chapter 14 of the LRM. IO characteristics are dictated by the underlying Unix IO subsystem which is bit-stream oriented for disc IO and block oriented for tape IO. Other devices have additional characteristics.

Ada Sequential_IO and Direct_IO is implemented such that reads and writes call the Unix system calls read(2) and write(2). If you instantiate Sequential_IO or Direct_IO with a constrained type, there is a one-to-one correspondence between Ada read/writes to UNIX read/writes. Otherwise, there are 2 Unix reads/writes to one Ada read/write, one for the size and one for the data.

12. Implementation Limits

The following limits are actually enforced by the implementation. It is not intended to imply that resources up to or even near these limits are available to every program.

Line Length

The implementation supports a maximum line length of 500 characters including the end of line character.

Record and Array Sizes

The maximum size of a statically sized array or record type is 4,000,000 x STORAGE_UNITS. A record type or array type declaration that exceeds these limits will generate a warning message.

Default Stack Size for Tasks

In the absence of an explicit STORAGE_SIZE length specification every task except the main program is allocated a fixed size stack of 10,240 STORAGE_UNITS. This is the value returned by T'STORAGE_SIZE for a task type T. The minimum stack size for tasks is 5120.

Default Collection Size

In the absence of an explicit STORAGE_SIZE length attribute the default collection size for an access type is 100 times the size of the designated type. This is the value returned by T'STORAGE_SIZE for an access type T.

Limit on Declared Objects

There is an absolute limit of 6,000,000 x STORAGE_UNITS for objects declared statically within a compilation unit. If this value is exceeded the compiler will terminate the compilation of the unit with a FATAL error message.